计算机学院专业课

计算机组成

机器语言(2)

高小鹏

北京航空航天大学计算机学院
系统结构研究所

Levels of Representation/Interpretation

High Level Language Program (e.g., C)

```
temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;
```

Compiler

Assembly Language Program (e.g., MIPS)

```
lw   $t0, 0($2)
lw   $t1, 4($2)
sw   $t1, 0($2)
sw   $t0, 4($2)
```

Anything can be represented as a *number*, i.e., data or instructions

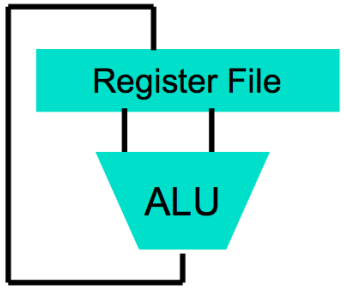
Assembler

Machine Language Program (MIPS)

```
0000 1001 1100 0110 1010 1111 0101 1000
1010 1111 0101 1000 0000 1001 1100 0110
1100 0110 1010 1111 0101 1000 0000 1001
0101 1000 0000 1001 1100 0110 1010 1111
```

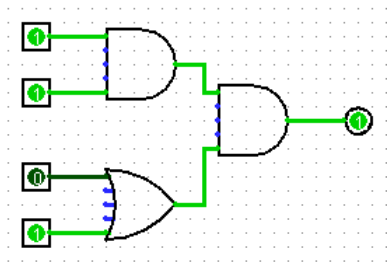
Machine Interpretation

Hardware Architecture Description (e.g., block diagrams)



Architecture Implementation

Logic Circuit Description (Circuit Schematic Diagrams)



提纲

- 内容主要取材：CS61C的6讲
  - http://inst.eecs.berkeley.edu/~cs61c/su12
- 不等式
- 伪指令
- 实现函数
- 函数调用约定

不等式

- 不等关系：<，<=，>，>=
  - MIPS：用增加1条额外的指令来支持所有的比较
- Set on Less Than (slt)
  - `slt dst,src1,src2`
  - 如果src1<src2，dst写入1，否则写入
- 与bne, beq, and $0组合即可实现所有的比较

不等式

- 实现<

C代码	MIPS汇编

if (a < b) {
... /* then */
}
(假设：a→$s0,b→$s1)
	slt $t0,$s0,$s1
	# $t0=1 if a<b
	# $t0=0 if a>=b
	bne $t0, $0,then
	# go to then
	# if $t0≠0

不等式

- 实现>=

C代码	MIPS汇编

if (a >= b) {
... /* then */
}
(假设：a→$s0,b→$s1)
	slt $t0,$s0,$s1
	# $t0=1 if a<b
	# $t0=0 if a>=b
	beq $t0, $0,then
	# go to then
	# if $t0=0

- Q：请自行完成
  - 交换src1与src2
  - 分别用beq与bne

不等式

- slt的3种变形
  - sltu dst,src1,src2: 无符号数比较
  - slti dst,src,imm: 与常量比较
  - sltiu dst,src,imm: 与无符号常量比较
- 示例：

```
addi  $s0,$0,-1   # $s0=0xFFFFFFFF
slti  $t0,$s0,1   # $t0=1
sltiu $t1,$s0,1   # $t1=0
```

不等式

- 用slt实现<，>，<=，>=
  - a<b：直接运用slt指令即可
  - a>b：由于a>b与b<a完全是相同的，因此可以改为用slt判断b<a
  - a≤b：由于a≤b与$\overline{b < a}$是等价的，因此将slt判断b<a的结果取反即可
    - “slt 结果，b，a”执行结果为0，则原条件为真
  - a≥b：方法同上

原条件	等价条件	指令用法	结果寄存器的0/1值含义
a < b		slt 结果寄存器, a, b	0：原条件为假 1：原条件为真
a > b	b < a	slt 结果寄存器, b, a	0：原条件为假 1：原条件为真
a <= b	$\overline{b < a}$	slt 结果寄存器, b, a	0：原条件为真 1：原条件为假
a >= b	$\overline{a < b}$	slt 结果寄存器, a, b	0：原条件为真 1：原条件为假

[AD]MIPS的Signed与Unsigned

- 术语Signed与Unsigned有3种不同含义
  - 位扩展
    - `lb`：扩展
    - `lbu`：无扩展
  - 溢出检测
    - `add, addi, sub, mult, div`：检测
    - `addu, addiu, subu, multu, divu`
  - 符号数
    - **`slt, slti`：符号数**
    - **`sltu, sltiu`：无符号数**

Question: What C code properly fills in the following blank?

```
do {i--;} while(__________);
```

```
Loop:                    # i→$s0, j→$s1
addi $s0,$s0,-1          # i = i - 1
slti $t0,$s1,2           # $t0 = (j < 2)
beq  $t0,$0 ,Loop        # goto Loop if $t0==0
slt  $t0,$s1,$s0         # $t0 = (j < i)
bne  $t0,$0 ,Loop        # goto Loop if $t0!=0
```

- ☐ `j ≥ 2 || j < i`
- ☐ `j ≥ 2 && j < i`
- ☐ `j < 2 || j ≥ i`
- ☐ `j < 2 && j ≥ i`

提纲

- 内容主要取材：CS61C的6讲
  - http://inst.eecs.berkeley.edu/~cs61c/su12
- 不等式
- 伪指令
- 实现函数
- 函数调用约定

伪指令

- 很多C语句对应到MIPS指令时很不直观
  - 例如：C的赋值语句a=b，会用某条运算指令实现，如addi
- MIPS定义了一组 伪指令，从而使得程序更可读更易编写
  - 伪指令不是真正的指令
  - 伪指令只是增加了可读性
  - 伪指令要被转换为实际指令
- 示例

```
move dst,src     转换为
addi dst,src,0
```

常用伪指令

- Move
  - `move dst, src`
  - 把`src`赋值给`dst`
- Load Address (`la`)
  - `la dst,label`
  - 加载特定标号对应的地址至`dst`
- Load Immediate (`li`)
  - `li dst,imm`
  - 加载一个32位立即数至`dst`

汇编寄存器

- 问题
  - 汇编器把一条伪指令转换为真实指令时，可能需要多条真实指令
  - 这组真实指令之间就必须通过某个寄存器来传递信息
  - 如果任意使用某个寄存器，则存在这个寄存器被汇编器误写的可能
- 解决方案
  - 保留$1（$at）作为汇编器专用寄存器
  - 由于汇编器会使用这个寄存器，因此从代码安全角度，其他代码不应再使用这个寄存器

编号	名称	用途
0	$zero	常量0
1	$at	汇编器保留
2-3		
4-7		
8-15	$t0-$t7	临时变量
16-23	$s0-$s7	程序变量
24-25	$t8-$t9	临时变量
28		
29		
30		
31		

MAL vs. TAL

- True Assembly Language (TAL)
  - 真实指令，是计算机能够理解和执行的
- MIPS Assembly Language (MAL)
  - 提供给汇编程序员使用的指令（包含伪指令）
  - 每条MAL指令对应1条或多条TAL指令
    - 主要是针对伪指令
- TAL ⊂ MAL

提纲

- 内容主要取材：CS61C的6讲
  - http://inst.eecs.berkeley.edu/~cs61c/su12
- 不等式
- 伪指令
- 实现函数
- 函数调用约定

实现函数的6个步骤

- 1、调用者把参数放置在某个地方以便函数能访问
- 2、调用者转移控制给被调用的函数
- 3、函数获取局部变量对应的空间
- 4、函数执行具体功能
- 5、函数把返回值放置在某个地方，然后恢复使用的资源
- 6、返回控制给调用者

函数相关寄存器

- 由于寄存器比主存快，因此尽可能的用寄存器
- $a0-$a3：4个传递参数的寄存器
- $v0-$v1：2个传递返回值的寄存器
- $ra：返回地址寄存器，保存着调用者的地址

编号	名称	用途
0	$zero	常量0
1	$at	汇编器保留
2-3	$v0-$v1	返回值
4-7	$a0-$a3	参数
8-15	$t0-$t7	临时变量
16-23	$s0-$s7	程序变量
24-25	$t8-$t9	临时变量
28		
29		
30		
31	$ra	返回地址

函数调用指令

- Jump and Link (`jal`)
  - `jal label`
  - 把jal的下一条指令的地址保存在`$ra`，然后跳转到`label`(即函数地址)
  - jal的用途是调用函数
- Jump Register (`jr`)
  - `jr src`
  - 无条件跳转到保存在`src`的地址(通常就是`$ra`)
  - jr的用途是从函数返回

PC

- PC（program counter）是一个特殊寄存器，用于保存当前正在指令的指令的地址
  - PC是冯式体系结构计算机的关键环节
  - 在MIPS中，PC对于程序员不可见，但可以被`jal`访问
- 注意：保存在`$ra`的是`PC+4`，而不是`PC`
  - 否则当函数返回的时候，就会再次返回到`jal`本身了

函数调用示例

```
... sum(a,b); ...          /* a→$s0,b→$s1 */

int sum(int x, int y) {
  return x+y;
}
```

C

MIPS

```
1000   addi $a0,$s0,0        # x = a
1004   addi $a1,$s1,0        # y = b
1008   jal  sum              # $ra=1012, goto sum
1012
...

2000   sum: add $v0,$a0,$a1
2004   jr   $ra              # return
```

地址

实现函数的6个步骤

- 1、调用者把参数放置在某个地方以便函数能访问
  - `$a0~$a3`
- 2、调用者转移控制给被调用的函数
  - `jal`
- 3、函数获取局部变量对应的空间？？
- 4、函数执行具体功能
- 5、函数把返回值放置在某个地方，然后恢复使用的资源
  - `$v0~$v1`
- 6、返回控制给调用者

保存和回复寄存器

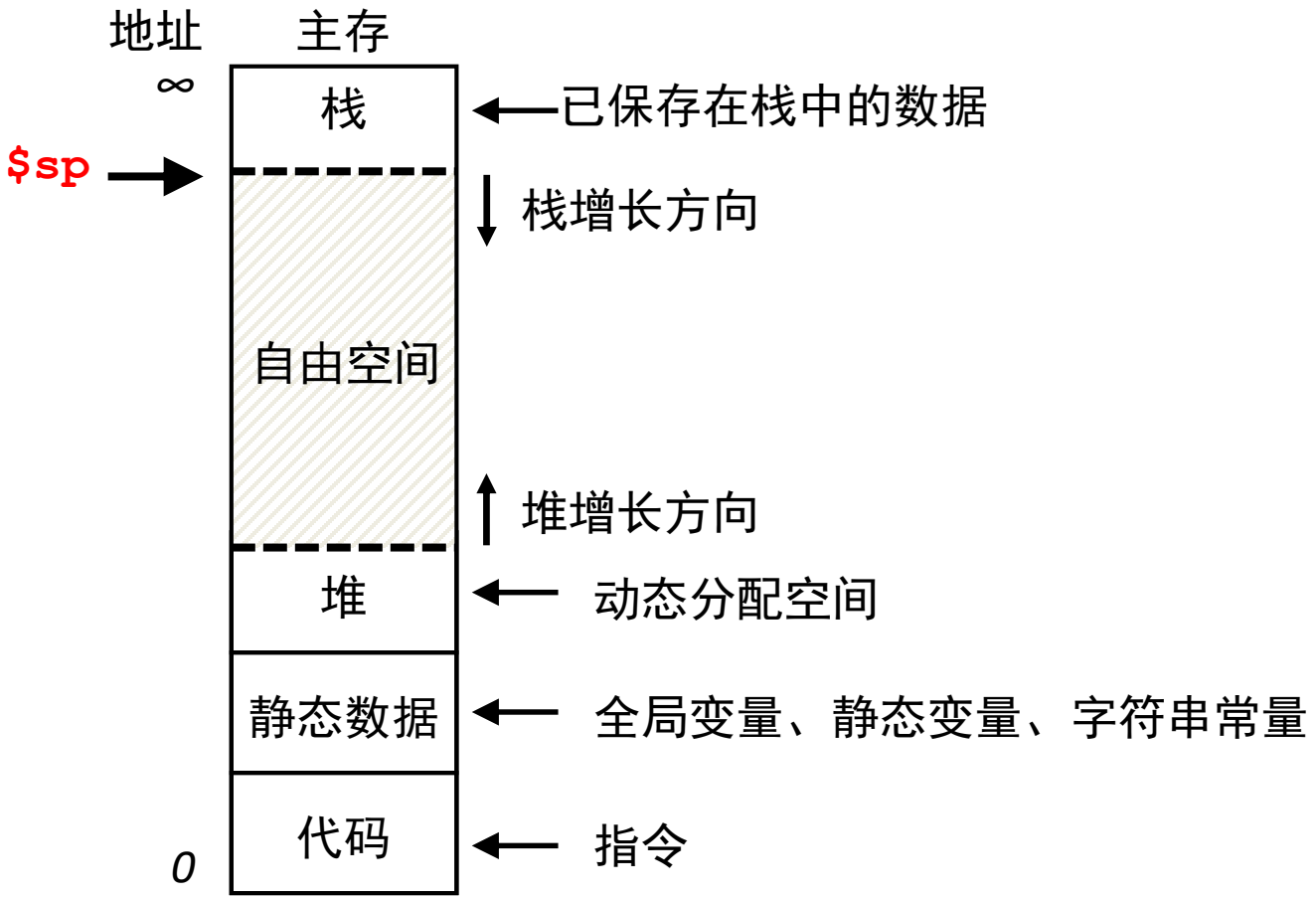
- Q：为什么需要保存寄存器？
  - 理由1：寄存器数量太少，不可能只用寄存器就能编写实用程序
  - 理由2：如果被调用函数继续调用函数，会发生什么？

    ($ra 被覆盖了！)
- Q：寄存器保存在什么地方？
  - 栈！
- $sp：栈指针寄存器。指针指向栈底

编号	名称	用途
0	$zero	常量0
1	$at	汇编器保留
2-3	$v0~$v1	返回值
4-7	$a0~$a4	参数
8-15	$t0-$t7	临时变量
16-23	$s0-$s7	程序变量
24-25	$t8-$t9	临时变量
28		
29	$sp	栈指针
30		
31	$ra	返回地址

主存分配的基本方案

- 栈帧：函数为获得保存寄存器而将$sp向0方向调整的空间
  - 栈帧容量 = 要保存的寄存器数量 X 4



函数示例

```
int sumSquare(int x, int y) {
   return mult(x,x)+ y; }
```

- 都需要保存哪些寄存器？
  - 1）$ra：由于sumSquare要调用mult，因此$ra会被覆盖
  - 2）$a1：给sumSquare传递y，但还给mult传递x
    - 关键是在sumSquare中，会先传递x然后才引用y，因此y就被x覆盖了
- 为了保存这2个寄存器，首先需要将$sp 向下移动8个字节（2个寄存器）
  - 这一新分配的空间被称为栈帧

函数示例

```
int sumSquare(int x, int y) {
   return mult(x,x)+ y; }
```

```
sumSquare:
        addi $sp,$sp,-8      # make space on stack
push    sw $ra, 4($sp)       # save ret addr
        sw $a1, 0($sp)       # save y
        move $a1,$a0         # set 2nd mult arg
        jal mult             # call mult
        lw $a1, 0($sp)       # restore y
        add $v0,$v0,$a1      # ret val = mult(x,x)+y
pop     lw $ra, 4($sp)       # restore ret addr
        addi $sp,$sp,8       # restore stack
        jr $ra
mult:   ...
```

函数的架构

- framesize = 要保存的寄存器个数 x 4

Prologue

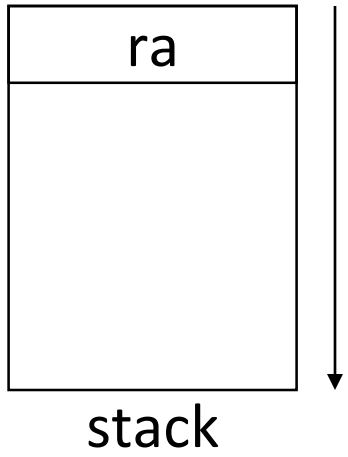
```
func_label:
addi $sp,$sp, -framesize
sw $ra, [framesize-4]($sp)
save other regs if need be
```

Body **(可以调用其他函数…)**

```
...
```

Epilogue

```
restore other regs if need be
lw $ra, [framesize-4]($sp)
addi $sp,$sp, framesize
jr $ra
```

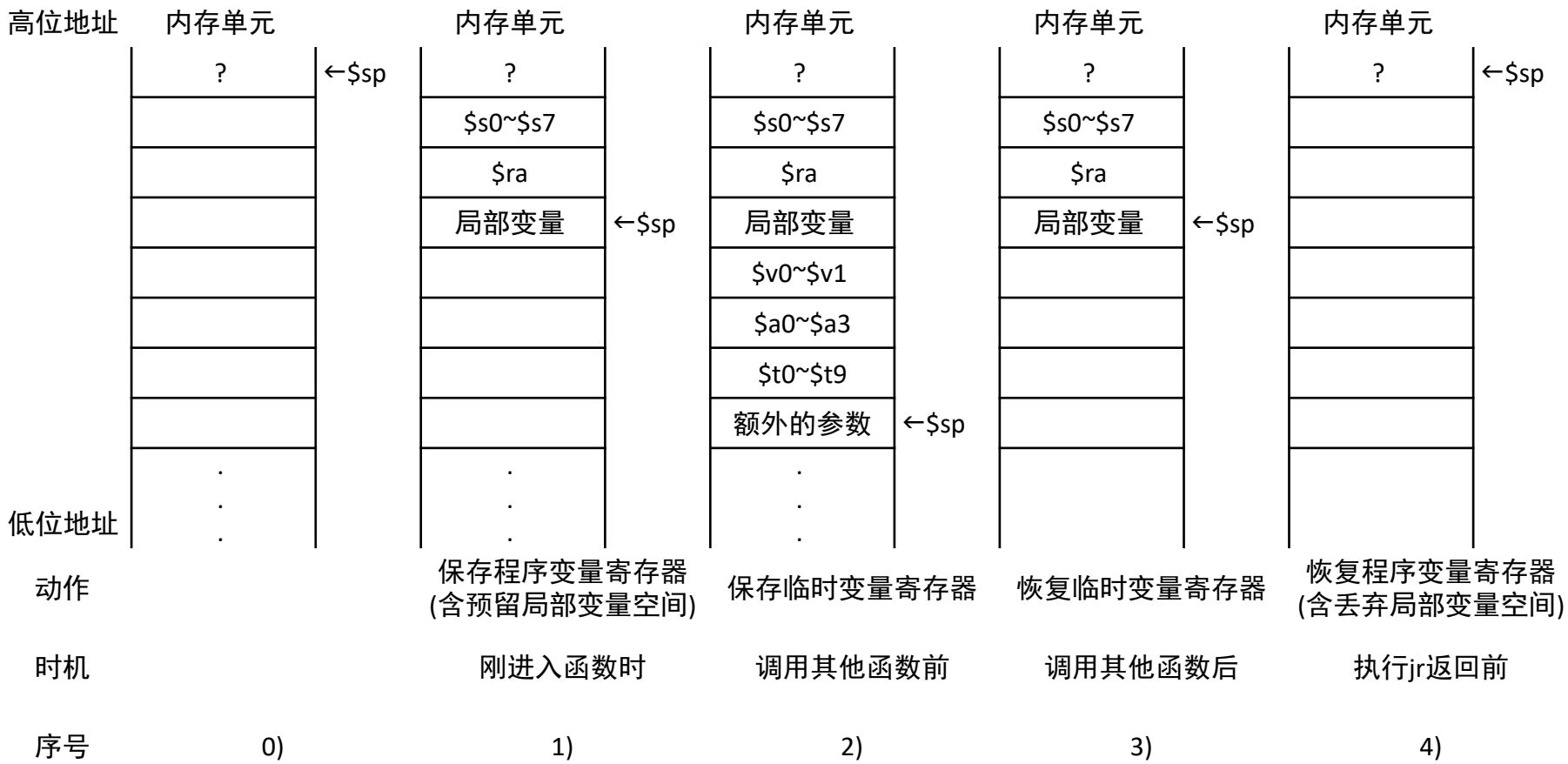


局部变量和数组

- 由于寄存器只有32个，因此编译器绝大多数情况下不可能把函数需要的所有局部变量都分配在寄存器
- 局部变量存放在函数自己的栈帧中
  - 这样当函数返回时，随着$sp的回调，局部变量自然就被释放了
- 其他局部变量，如数组、结构等，同样也是存储在栈帧中
- 为局部变量分配空间的方法是与保存寄存器是完全相同
  - 将$sp向下调整，产生的空间用于存储局部变量

局部变量和数组

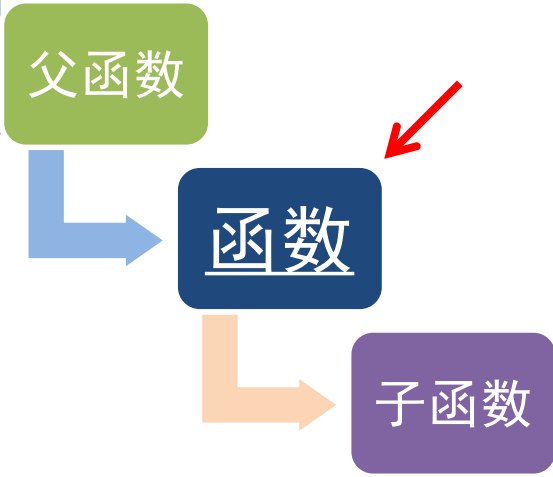
- 由于寄存器只有32个，因此编译器绝大多数情况下不可能把函数需要的所有局部变量都分配在寄存器



提纲

- 内容主要取材：CS61C的6讲
  - http://inst.eecs.berkeley.edu/~cs61c/su12
- 不等式
- 伪指令
- 实现函数
- 函数调用约定

寄存器的保护分析



- $s0~$s7：程序员变量
  - 分析：所有函数都要使用程序员变量
    - 程序员变量的生命周期与函数生命周期相同，即进入函数就有效，退出函数后无效
  - 保护前提：用哪些，保护哪些
  - 保护动作：刚进入函数时保存；退出函数前恢复
- $ra：函数返回地址
  - 保护前提：如果继续调用子函数，那么就必须保护
    - 否则函数就不能返回至父函数
  - 保护动作：刚进入函数时保存；退出函数前恢复

寄存器的保护分析

- $t0~$t9：临时变量（前提：MIPS约定其服务于表达式计算）
  - 无需保护：表达式中不调用子函数，故不存在被子函数修改的可能

```
e = a + b + c + d ;  // $t0 = a + b, $t1 = c + d
f1() ;
```

  - 需要保护：表达式中调用子函数，故存在被子函数修改的可能

```
z = x + y + f2() ;    // $t0 = x + y
                      // $t0有可能被f2()修改
```

  - 保护前提：如果其值在子函数调用前后必须保持一致，则：
  - 保护动作：调用子函数前保存；在调用子函数后恢复

寄存器的保护分析

- $a0~$a3，$v0~$v1：参数，返回值
  - $a0~$a3：传递给了子函数，故子函数可自由使用
  - $v0~$v1：因为子函数会设置返回值，故子函数可自由使用
  - 保护前提：如果其值在调用子函数前后必须保持一致，则：
  - 保护动作：调用子函数前保存；在调用子函数后恢复

- $at：汇编器使用，故无需保护
- $k0~$k1：操作系统使用（现阶段可以不使用）
- $sp：栈切换时才需要保护（现阶段可以不使用）
  - 栈切换：通常属于操作系统范畴
- $fp、$gp：在生成复杂的存储布局时使用（属于编译范畴）

寄存器的保护分析

序号	名称	MIPS汇编约定的用途	分类	保护的前提条件	何时保护与恢复
16~23	$s0~$s7	程序员变量	强保护	如果需要使用	进入函数时保存 退出函数前恢复
31	$ra	函数返回地址		如果调用子函数	
2~3	$v0~$v1	函数返回值	弱保护	如果要求其值在调用子函数前后必须保持一致	调用子函数前保存 子函数返回后恢复
4~7	$a0~$a3	函数参数			
8~15, 24~25	$t0~$t9	临时变量			

- Saved Register：$s0~$s7，$ra
  - 英文：preserved register；中文：保护寄存器，保存寄存器
- Volatile Register：$t0~$t9，$a0~$a3，$v0~$v1
  - 英文：non-preserved register；中文：非保护寄存器，非保存寄存器

强保护寄存器的保护机制代码框架

- 进入函数后，立刻调整栈并保存；在函数返回时才恢复

```
1  函数_lable :
2      addiu $sp, $sp, -framesize      分配栈帧
3      sw $ra, [framesize-4]($sp)      保存$ra
4      sw $s0, [framesize-8]($sp)      保存$s0~$s7中后续要使用的寄存器
5      sw $s1, [framesize-12]($sp)
6      # …
7      sw $s7, 0($sp)
8
9      # 使用$s0~$s7
10     # 调用其他子函数
11
12     lw $s7, 0($sp)                  恢复函数保护的寄存器
13     # …
14     lw $s1, [framesize-12]($sp)
15     lw $s0, [framesize-8]($sp)
16     lw $ra, [framesize-4]($sp)
17     addiu $sp, $sp, framesize       回收栈帧
18     jr $ra
```

弱保护寄存器的保护机制代码框架

- 在调用其他子函数时才调整栈并保存；在子函数返回时立即恢复
  - 有N次子函数调用，就可能N次保护与恢复；不调用子函数，就无需保护

```
1  函数_lable :
2      # 其他函数代码
3
4      addiu $sp, $sp, -tmpsize          在调用子函数前分配栈帧
5      sw $t[i], [tmpsize-4]($sp)        保存寄存器
6      sw $t[i+1], [tmpsize-8]($sp)
7      # 其他需保存的寄存器
8      sw $t7, 0($sp)
9
10     jal 子函数                        调用子函数
11
12     lw $t7, 0($sp)                    恢复寄存器
13     # …
14     lw $t[i+1], [tmpsize-8]($sp)
15     lw $t[i], [tmpsize-4] ($sp)
16     addiu $sp, $sp, tmpsize           回收栈帧
17
18     # 其他函数代码
```

Q：下列哪个陈述是错误的？

- MIPS使用`jal`指令来调用函数，并使用`jr`指令函数返回
- `jal`保存`PC+1`到`$ra`
- 函数如果不担心`$ti`值在调用子函数后发生改变，则无需保存和回复它们
- 函数如果要使用(`$si`)，就必须保存和回复它们

小节

- 通过组合beq与slt 指令，可以实现各种比较判断
- 伪指令使得代码更易读
- 函数机制
  - 用jal实现函数调用，用jr实现函数返回
  - 用$a0-$a3传递参数，用$v0-$v1传递返回值
  - 寄存器保护
    - 从用途看，寄存器可以分为强保护和弱保护两大类
    - 强保护寄存器：一进函数就保护，退出时恢复
    - 弱保护寄存器：调用子函数时保护，子函数一返回就恢复
  - 栈用于保存寄存器、局部变量等

作业

- 《计算机组成与设计》
  - WORD：2.15，2.19
- 把1～100进行倒序
  - 直接在汇编的data段中定义100个字节，依次存放1，2，。。。。，100
  - 倒序，即结果为100，99，。。。1
  - 提交asm和报告
    - 报告：分析你的asm总共需要执行的指令数（不能只给结果，需要分析和计算的过程）